BH01783

DATA STRUCTURE AND ALGORITHMS

Create by Kim Tien Dat

MY TEAM

Kim Tien Dat

BH01783

CONTENT

INTRODUCTION

Overview of data structures and algorithms

Focus on efficiency, performance, and problem-solving

Importance in software development and programming

DATA STRUCTURES AND COMPLEXITY

Common Data Structures

- Stack: A linear data structure that follows the Last In First Out (LIFO) principle. Key operations are Push (add an element) and Pop (remove the last element). Stacks are used in recursion, memory management, and parsing.
- Queue: Follows the First In First Out (FIFO) principle, where elements are added at the back and removed from the front. Used in task scheduling, buffering, and breadth-first search.
- Linked List: A sequence of nodes where each node points to the next, allowing dynamic memory allocation. Used for implementing stacks, queues, and managing memory more flexibly than arrays.

DATA STRUCTURES AND COMPLEXITY

Key Operations

- Each data structure has primary operations which determine its usage:
- Insert: Adding new elements to the structure.
- Delete: Removing elements from the structure.
- Search: Finding specific elements within the structure.
- Each of these operations can vary in efficiency depending on the data structure used.

DATA STRUCTURES AND COMPLEXITY

Complexity: Time and Space Analysis

- Time Complexity: Measures the time required for operations as the data size grows, commonly expressed as Big O notation (e.g., O(1), O(n), O(log n)).
- O(1): Constant time (e.g., accessing an array element by index)
- O(n): Linear time (e.g., traversing a linked list)
- O(log n): Logarithmic time (e.g., binary search)
- Space Complexity: Amount of memory needed for a data structure.
- Efficient memory use is essential for large data sets, especially in linked structures where overhead can vary.

DATA STRUCTURES AND COMPLEXITY

Choosing the Right Data Structure

- Performance of operations (e.g., insertion, deletion, lookup) is crucial for selecting the appropriate data structure for specific use cases.
- Example: Use a stack when last-in-first-out operations are required, and a queue for first-in-first-out processing.

MEMORY STACK

Definition: Last In, First Out (LIFO) Structure

- The Memory Stack is a structure where data is added and removed in a Last In, First Out (LIFO) order.
- Used by programming languages to manage function calls, local variables, and other temporary data within a controlled memory space.

- Push: Adds an item to the top of the stack.
- Pop: Removes the top item from the stack.
- Peek: Views the top item without removing it.
- IsEmpty: Checks if the stack is empty.

Key Operations

MEMORY STACK

Stack Frames in Function Calls

- Each function call creates a stack frame in memory, containing the function’s parameters, local variables, and return address.
- When a function is called, a new stack frame is “pushed” onto the memory stack. When the function completes, its frame is “popped” off.
- Stack frames allow programs to execute recursively and maintain separate contexts for each function call.

- Memory Management: Controls memory for short-lived data, automatically freeing it when a function completes.
- Function Execution: Tracks the order of function calls, enabling recursion and nested function calls.
- Error Detection: Detects stack overflow, which occurs if there are too many function calls (often due to infinite recursion).

Importance of the Memory Stack

QUEUE (FIFO)

Key Operations

A Queue is a linear data structure that operates on the First In, First Out (FIFO) principle, meaning the first element added is the first one removed.

This structure is used in situations where order must be preserved, such as processing tasks sequentially.

QUEUE (FIFO)

Definition: First In, First Out (FIFO) Structure

- Enqueue: Adds an element to the end of the queue.
- Dequeue: Removes the element at the front of the queue.
- Peek/Front: Views the element at the front without removing it.
- IsEmpty: Checks if the queue has no elements

QUEUE (FIFO)

Implementations of Queue

- Array-Based Implementation: Stores queue elements in a fixed-size array. Simpler but has a fixed limit unless a circular queue approach is used to wrap around and optimize space.
- Linked List-Based Implementation: Dynamically allocated memory allows queues to grow or shrink as needed. Preferred when the size of the queue isn’t known in advance.

QUEUE (FIFO)

Real-World Examples

- Task Scheduling: In operating systems, processes or tasks are placed in a queue to ensure each task gets processed in the order it arrived.
- Printer Queue: Print jobs are managed in a queue, ensuring that the first job sent to the printer is completed first.
- Breadth-First Search (BFS): Queue is used to explore nodes level by level in graph traversal, helping to maintain the order of nodes to visit next.

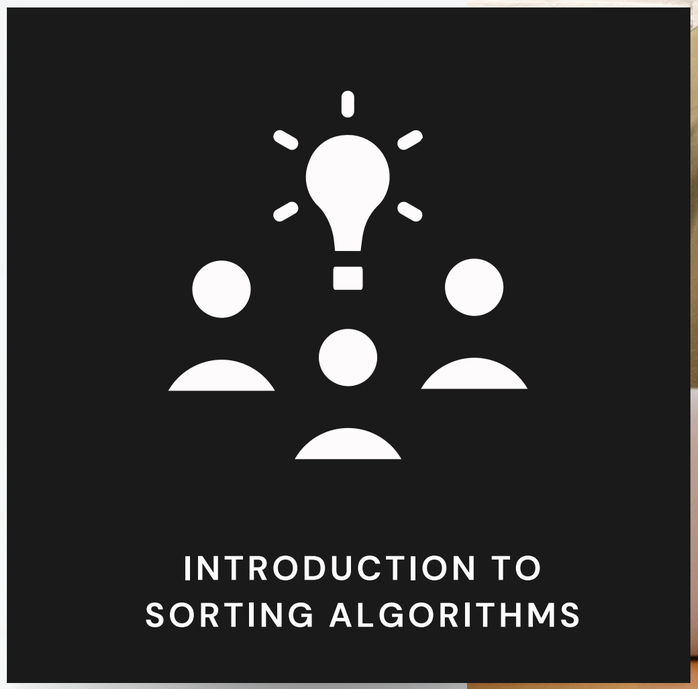
QUEUE (FIFO)

Advantages of FIFO

- Predictability: FIFO order is predictable, which is crucial for task management, where each item is processed in a fair, ordered sequence.
- Efficiency in Resource Management: Queueing systems ensure resources are distributed based on the order requests are received, useful in network data packets, customer support lines, etc.

SORTING ALGORITHMS

- Sorting algorithms arrange data in a specific order, typically ascending or descending.
- Choosing the right algorithm depends on factors like data size, memory limitations, and performance requirements.



SORTING ALGORITHMS

Merge Sort:
Divide and Conquer algorithm that splits the data into halves, recursively sorts each half, and then merges the sorted halves.
Time Complexity: O(n log n) consistently across best, average, and worst cases.
Space Complexity: Requires additional space (O(n)) for merging steps.
Stability: Stable sort (preserves the order of equal elements).

COMPARISON OF TWO SORTING ALGORITHMS (E.G., MERGE SORT VS. QUICK SORT)

Quick Sort:
Also a Divide and Conquer algorithm, but partitions the data based on a pivot, then recursively sorts elements on either side of the pivot.
Time Complexity: Best and average cases are O(n log n), but the worst case (already sorted data or bad pivot choices) is $O(n^2)$.
Space Complexity: O(log n) on average, generally less than Merge Sort.
Stability: Not stable by default.

SORTING ALGORITHMS

- Sorting algorithms are commonly compared using Big O notation:
- O(n log n) for efficient sorts like Merge Sort and Quick Sort.
- O(n²) for simpler, less efficient sorts like Bubble Sort and Selection Sort.
- Time complexity impacts scalability: O(n log n) algorithms are preferred for larger data sets.

TIME COMPLEXITY ANALYSIS

SORTING ALGORITHMS

- Merge Sort requires extra space, making it less suitable for memory-constrained systems.
- Quick Sort is generally more space-efficient since it sorts in place, though some memory is required for recursion.

SPACE COMPLEXITY ANALYSIS

SORTING ALGORITHMS

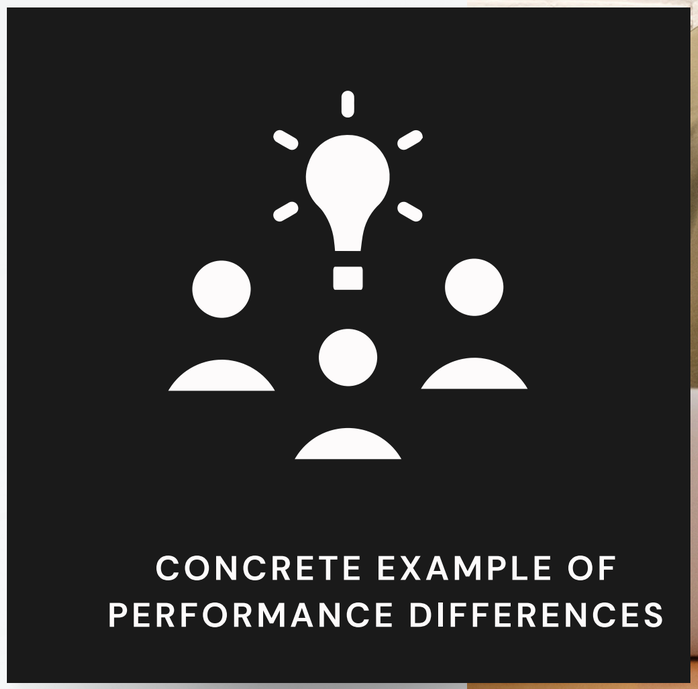
- Stable algorithms maintain the relative order of equal elements, useful for cases where duplicate elements are meaningful (e.g., sorting records by date while maintaining name order).
- Merge Sort is stable; Quick Sort is not, unless specifically modified.

SORTING ALGORITHMS

- Merge Sort: Preferred for large data sets or linked lists where stability is essential.
- Quick Sort: Often faster in practice due to less overhead and is widely used in systems with constrained memory or where stability is not critical.

SORTING ALGORITHMS

- For a nearly sorted list, Quick Sort may perform poorly if pivots are not well-chosen, leading to $O(n^2)$ time.
- Merge Sort, however, consistently performs at $O(n \log n)$, making it reliable for diverse cases, though it has higher space requirements.



CONCRETE EXAMPLE OF PERFORMANCE DIFFERENCES

NETWORK SHORTEST PATH ALGORITHMS

INTRODUCTION TO SHORTEST PATH ALGORITHMS

- Shortest path algorithms determine the minimum path between nodes in a network or graph, optimizing for distance, time, or cost.
- These algorithms are widely used in GPS navigation, network routing, and social network analysis.

NETWORK SHORTEST PATH ALGORITHMS

COMPARISON OF TWO ALGORITHMS: DIJKSTRA'S ALGORITHM VS. PRIM'S ALGORITHM

- Dijkstra's Algorithm:
- Purpose: Finds the shortest path from a single source node to all other nodes in a graph with non-negative weights.
- How it Works: Uses a priority queue to expand the shortest path node by node, updating distances to neighboring nodes.
- Time Complexity: $O(V^2)$ with a simple implementation, or $O(E + V \log V)$ using a priority queue.
- Best Used For: Dense graphs or when we only need shortest paths from one starting node.

NETWORK SHORTEST PATH ALGORITHMS

COMPARISON OF TWO ALGORITHMS: DIJKSTRA'S ALGORITHM VS. PRIM'S ALGORITHM

- Prim's Algorithm (Prim-Jarnik):
- Purpose: Constructs a minimum spanning tree (MST), connecting all nodes with the minimum total edge weight.
- How it Works: Starts from an initial node and adds edges with the lowest weight until all nodes are connected.
- Time Complexity: O(E log V) with a priority queue or $O(V^2)$ in simpler implementations.
- Best Used For: Finding an MST in weighted graphs, typically for optimizing networks without specific start or end nodes (e.g., connecting network cables).

NETWORK SHORTEST PATH ALGORITHMS

PERFORMANCE ANALYSIS

- Dijkstra’s Algorithm is efficient for finding shortest paths from one node, especially with a priority queue. It’s commonly used in routing protocols like OSPF (Open Shortest Path First).
- Prim’s Algorithm is designed for minimum spanning trees, ensuring all nodes are connected at minimal cost, making it useful for network design.

NETWORK SHORTEST PATH ALGORITHMS

PRACTICAL EXAMPLES

- Dijkstra’s Algorithm: Used in GPS systems to calculate the fastest route from a start point to a destination.
- Prim’s Algorithm: Useful in infrastructure planning, such as laying out fiber-optic cables, where all locations must be connected with minimal wiring costs.

NETWORK SHORTEST PATH ALGORITHMS

KEY DIFFERENCES

- Purpose: Dijkstra’s focuses on the shortest path to specific nodes; Prim’s connects all nodes with minimal edge weight.
- Type of Graph: Dijkstra’s requires non-negative edge weights, while Prim’s can be used on any weighted undirected graph.
- Output: Dijkstra’s provides the shortest path tree from a source, while Prim’s gives a minimum spanning tree.

STACK ABSTRACT DATA TYPE (ADT)

- A Stack is an Abstract Data Type (ADT) that follows the Last In, First Out (LIFO) principle, where the last element added is the first one to be removed.
- It is characterized by two main operations: push (adding an element) and pop (removing the top element).

DEFINITION OF STACK ADT

The stack can be implemented using two common structures:

- Array: A fixed-size collection of elements, where the top of the stack is tracked by an index. This implementation is straightforward but has a maximum capacity defined at creation.
- Linked List: A dynamic structure where each element (node) points to the next one. The stack can grow or shrink in size, making it more flexible than an array.

STACK DATA STRUCTURE

STACK ABSTRACT DATA TYPE (ADT)

STACK OPERATIONS

Initialize the Stack:
Create an empty stack with a specific structure (array or linked list).

Push Operation:
Definition: Adds an element to the top of the stack.

Pop Operation:
Definition: Removes the element from the top of the stack and returns it.

Peek Operation:
Definition: Returns the top element without removing it from the stack.

Check if the Stack is Empty:
Definition: Determines if there are any elements in the stack.

STACK ABSTRACT DATA TYPE (ADT)

STACK OPERATIONS

Push Operation:
Definition: Adds an element to the top of the stack.

```
public void push(int value) {
    if (top >= stack.length - 1) {
        System.out.println("Stack Overflow");
    } else {
        stack[++top] = value;
    }
}
```

STACK ABSTRACT DATA TYPE (ADT)

STACK OPERATIONS

Pop Operation:
Definition: Removes the element from the top of the stack and returns it.

```
public int pop() {
    if (top < 0) {
        System.out.println("Stack Underflow");
        return -1; // Or handle underflow appropriately
    } else {
        return stack[top--];
    }
}
```

STACK ABSTRACT DATA TYPE (ADT)

STACK OPERATIONS

Peek Operation:
Definition: Returns the top element without removing it from the stack.

```
public int peek() {
    if (top < 0) {
        System.out.println("Stack is Empty");
        return -1;
    } else {
        return stack[top];
    }
}
```

STACK ABSTRACT DATA TYPE (ADT)

STACK OPERATIONS

Check if the Stack is Empty:
Definition: Determines if there are any elements in the stack.

```
public boolean isEmpty() {
    return top < 0;
}
```

STACK ABSTRACT DATA TYPE (ADT)

FULL SOURCE CODE FOR STACK IMPLEMENTATIONS

Array-Based Implementation:

```
public class Stack {
    private int maxSize;
    private int[] stack;
    private int top;

    public Stack(int size) {
        maxSize = size;
        stack = new int[maxSize];
        top = -1;
    }

    // Push, pop, peek, isEmpty methods here...
}
```

STACK ABSTRACT DATA TYPE (ADT)

FULL SOURCE CODE FOR STACK IMPLEMENTATIONS

Linked List-Based Implementation:

```
class Node {
    int data;
    Node next;
    public Node(int data) {
        this.data = data;
    }
}

public class Stack {
    private Node top;

    public void push(int value) {
        Node newNode = new Node(value);
        newNode.next = top;
        top = newNode;
    }

    public int pop() {
        if (top == null) throw new EmptyStackException();
        int value = top.data;
        top = top.next;
        return value;
    }

    // Peek and isEmpty methods here...
}
```

STACK ABSTRACT DATA TYPE (ADT)

- Array Implementation:
- Advantages: Simpler to implement and faster access to elements due to contiguous memory allocation.
- Disadvantages: Fixed size; can lead to overflow if not managed properly.
- Linked List Implementation:
- Advantages: Dynamic size; no overflow as it grows with the number of elements.
- Disadvantages: More overhead due to extra memory for pointers; potentially slower access due to non-contiguous memory allocation.

COMPARING ARRAY AND LINKED LIST IMPLEMENTATIONS

- Function Call Management: Stacks are used to keep track of function calls and local variables in programming languages (call stack).
- Undo Mechanisms: Many applications use stacks to manage undo operations (e.g., text editors).
- Expression Evaluation: Used in parsing expressions and implementing algorithms like postfix notation evaluation.

APPLICATIONS OF STACK ADT

ENCAPSULATION AND INFORMATION HIDING

- Definition: Encapsulation is an object-oriented programming principle that restricts access to certain components of an object and bundles the data (attributes) and methods (functions) that operate on the data into a single unit, typically a class.
- Purpose: It ensures that the internal representation of an object is hidden from the outside, exposing only what is necessary through a public interface.

WHAT IS ENCAPSULATION?

- Data Protection: By hiding internal data, encapsulation protects an object's state from unintended interference and misuse, preventing direct access to sensitive information.
- Modularity and Maintainability: Encapsulation promotes modularity, making it easier to change the internal implementation of a class without affecting external code that uses it. This leads to easier maintenance and reduced risk of introducing bugs.
- Code Reusability: Well-encapsulated classes can be reused across different programs, as they expose only the necessary functionality while keeping their inner workings hidden.

ADVANTAGES OF ENCAPSULATION

- Definition: Information hiding is the practice of restricting access to certain details of an object's implementation. It focuses on exposing only the essential features while concealing complex implementations and internal workings.
- Relation to Encapsulation: While encapsulation is about bundling data and methods, information hiding emphasizes hiding implementation details from the user.

WHAT IS INFORMATION HIDING?

- Abstraction: Simplifies the interface presented to users. Users interact with an ADT through its public methods without needing to understand its implementation.
- Reduced Complexity: By exposing only necessary functionalities, developers face less complexity, allowing them to focus on higher-level programming rather than getting bogged down by the underlying code.
- Improved Security: Limits the exposure of sensitive data and functions, reducing the risk of unauthorized access or unintended modifications.

BENEFITS OF INFORMATION HIDING

ENCAPSULATION AND INFORMATION HIDING

Example of Encapsulation

```java
public class BankAccount {
    // Private fields to store account details
    private String accountNumber;
    private double balance;

    // Constructor to initialize the bank account
    public BankAccount(String accountNumber, double initialBalance) {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }

    // Public method to deposit money
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    // Public method to withdraw money
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }

    // Public method to get the current balance
    public double getBalance() {
        return balance;
    }
}
```

IMPERATIVE ADTS AND OBJECT ORIENTATION

What are Imperative Abstract Data Types (ADTs)?

Definition: Imperative ADTs are data structures defined by their behavior and the operations that can be performed on them. They specify how data can be manipulated and how it interacts with the program.

Characteristics:
Focus on how data is stored and manipulated through procedures (functions or methods).
Operations on data are usually written as explicit sequences of commands or statements.

IMPERATIVE ADTS AND OBJECT ORIENTATION

Core Concepts of Imperative ADTs

- Encapsulation: Imperative ADTs encapsulate data and the operations that manipulate this data. This means the internal representation is hidden, and access is only allowed through defined methods.
- State and Behavior: An imperative ADT maintains its state (data) and provides behaviors (methods) that can modify that state.

IMPERATIVE ADTS AND OBJECT ORIENTATION

Transition to Object-Oriented Programming (OOP)

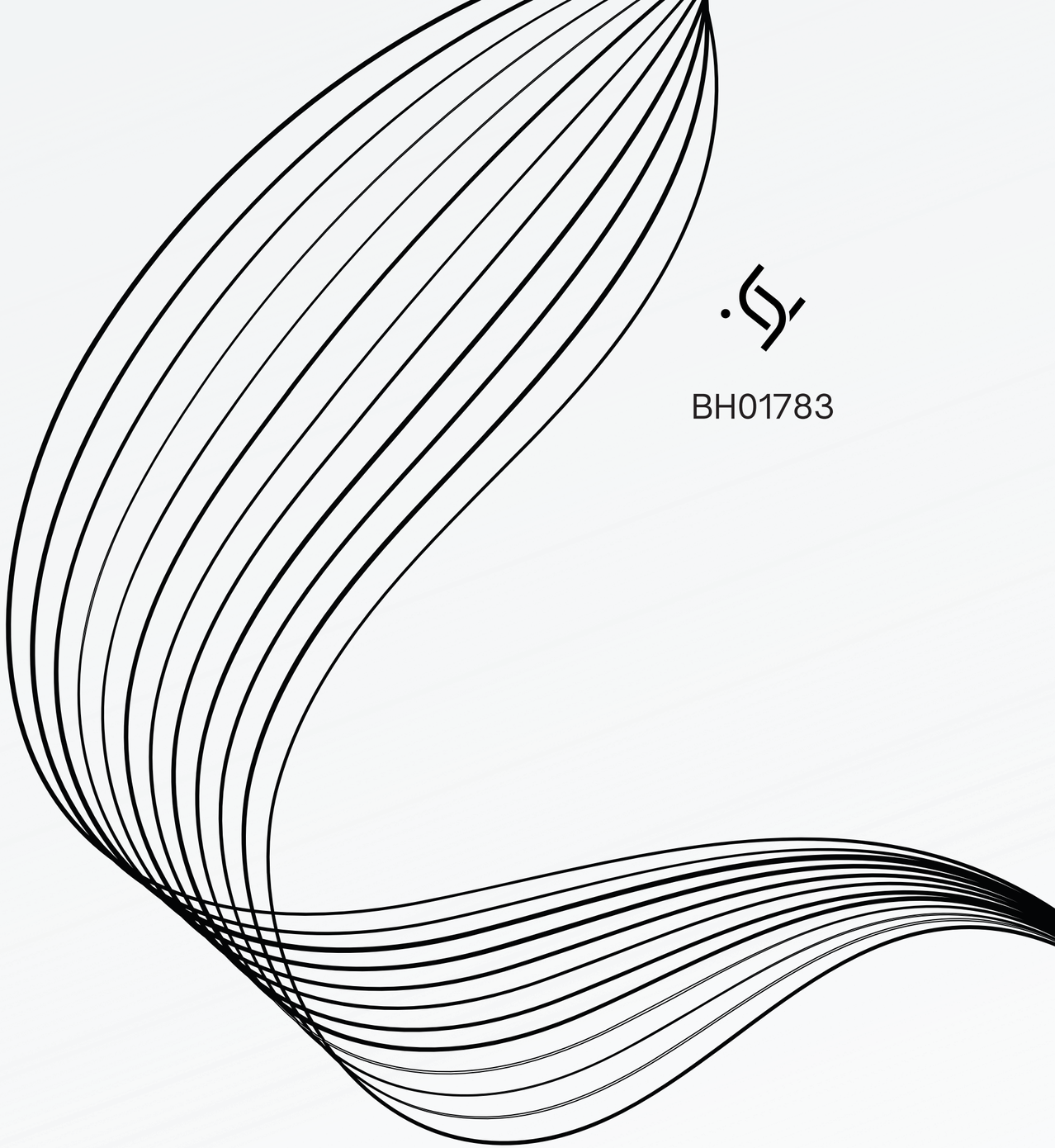
- Object Orientation: OOP extends the concepts of imperative ADTs by organizing code into classes and objects.
- Key Principles:
- Encapsulation: Continues to be important, where data and methods are bundled into classes.
- Inheritance: New classes can be created based on existing ones, allowing for code reuse and extending functionalities.
- Polymorphism: Objects can be treated as instances of their parent class, allowing for methods to be called on objects of different classes that share a common interface.

IMPERATIVE ADTS AND OBJECT ORIENTATION

Justification for the View

- Encapsulation and Information Hiding:
- Imperative ADTs serve as the basis for encapsulation in OOP, ensuring data integrity and protecting the internal state of objects.
- Modularity and Reusability:
- Both paradigms emphasize creating modular code that can be reused across applications. In OOP, classes serve as modules that can be instantiated and reused, aligning with the goals of imperative ADTs.
- Procedural Approach:
- Imperative programming focuses on procedures (functions) that operate on data, similar to how methods operate on objects in OOP. This procedural foundation is a precursor to the more structured approach of OOP.
- Language Transition:
- Many modern programming languages, like Java and C#, have incorporated both imperative and object-oriented features, allowing developers to define ADTs as classes while providing imperative control structures.

THANK'S FOR WATCHING



BH01783